Protein Sequence Database

A protein is a large molecule manufactured in the cell of a living organism to carry out essential functions within the cell. The primary structure of a protein is a sequence of amino acids. There are 20 common amino acids, each of which has a chemical name (e.g., "Glycine"), a three-letter abbreviation (e.g., "Gly"), and a one-letter code (e.g., "G"). See

`http://www.chemie.fu-berlin.de/chemistry/bio/amino-acids_en.html`

for a table about the chemistry of the amino acids and

`http://courses.cs.vt.edu/~algnbio/genetic_code/index.php`

for information about how the amino acids fit into the genetic code. The twenty one-letter codes are:

```
A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y
```

For the purpose of representing and manipulating the primary structure of a protein, it suffices to use the one-letter codes in a string. For example,

```
MLQSIIKNIWIPMKPYYTKVYQEIWIGMGLMGFIVYKIRAADKRSKALKASAPAPGHH
```

is the amino acid sequence for a human protein called "6.8 kDa mitochondrial proteolipid". In this project, amino acid sequences will always be upper-case, with no white space inserted. There are many online databases from which protein sequences can be obtained. One is SWISS-PROT, which can be found at `http://ca.expasy.org/sprot/`. As of March 27, 2004, SWISS-PROT contained database entries for 146,720 amino acid sequences. Each database entry has much more information about a protein than its sequence, as can be seen by going to

`http://ca.expasy.org/cgi-bin/get-full-entry?[SWISS_PROT-ID:'68MP_HUMAN']`

This is the entry for the protein whose sequence was given above.

A complete protein record may contain a fairly large number of logical fields. These are flagged with two-character sequences occurring at the beginning of each line. A full listing of the possible fields is given in Table 1 on page 2 of this specification. It is important to note that some protein records will contain only a proper subset of the possible fields. In addition the amount of data for each field can vary considerably.

For our purposes in this assignment, we will use a text file of modified, shortened SWISS-PROT entries. You do not need to be concerned with validating the correctness of the database entries.

A full description of the logical significance of the various fields, and any format constraints, is given in the UniProt User Manual, which is available at `http://us.expasy.org/sprot/userman.html`. Table 1 below describes the fields that are present in the shortened records we will be using. Figure 1 below shows a sample shortened protein record.

Note: some of the sequence data files may contain multiple entries corresponding to the same accession code. In such a case, your implementation should recognize if an accession code is already in the index structure, and if so simply reject the duplicate entries.

Table 1 Concise protein record field specifications:

Each line begins with a two-character line code, which indicates the type of data contained in the line. The line code is always followed by exactly three spaces. The line types and line codes that may appear in a concise entry, are shown in the table below.

Line code	Content	Occurrence in an entry	Comments
AC	Accession number(s)	Once	[O,P,Q][0-9][A-Z,0-9]^3[0-9]
DE	Description	Once or more	
OS	Organism species	Once or more	one organism per occurrence
OG	Organelle	Optional, zero or more	
CC	Comments or notes	Optional, zero or more	
KW	Keywords	Optional, zero or more	one or more keyword values per occurrence
SQ	Sequence data	Once	
//	Termination line	Once; ends the entry	

As shown in the table, some line types are found in all entries, others are optional. Some line types occur many times in a single entry. Each entry ends with a terminator line (`//`).

Note that some formatting details must be inferred from the sample data files provided on the course website, and the detailed documentation available online in the UniProt User Manual. Note that there are absolutely no stated limits on the lengths of the strings that occur in the protein records.

Figure 1 Sample Concise Protein Record:

```
AC   P10529
DE   Alpha-amylase A precursor (EC 3.2.1.1) (Taka-amylase A) (TAA) (1,4-alpha-D-
glucan glucanohydrolase).
OS   Aspergillus oryzae
KW   Carbohydrate metabolism; Hydrolase; Glycosidase; Calcium-binding;
KW   Signal; Glycoprotein; Multigene family; 3D-structure.
CC   -!- CATALYTIC ACTIVITY: Endohydrolysis of 1,4-alpha-glucosidic
CC   linkages in oligosaccharides and polysaccharides.
CC   -!- COFACTOR: Binds 2 calcium ions per subunit. Calcium is inhibitory
CC   at high concentrations.
CC   -!- SUBUNIT: Monomer.
CC   -!- BIOTECHNOLOGY: Used in the brewing industry to increase the
CC   fermentability of beer worts (including those made from unmalted
CC   cereals), in the starch industry to make high maltose and high DE
CC   syrups (starch saccharification), in the alcohol industry to
CC   reduce fermentation time, in the cereal food industry for flour
CC   supplementation and improvement of chilled and frozen dough, and
CC   in the forestry industry for low-temperature modification of
CC   starch. Sold under the name Fungamyl by Novozymes.
CC   -!- MISCELLANEOUS: The sequence of AMY1 and AMY2 is shown.
CC   -!- SIMILARITY: Belongs to family 13 of glycosyl hydrolases.
CC   ---------------------------------------------------------------------
CC   This SWISS-PROT entry is copyright. It is produced through a collaboration
CC   between  the Swiss Institute of Bioinformatics  and the  EMBL outstation -
CC   the European Bioinformatics Institute.  There are no  restrictions on  its
CC   use  by  non-profit  institutions as long  as its content  is  in  no  way
CC   modified and this statement is not removed.  Usage  by  and for commercial
CC   entities requires a license agreement (See http://www.isb-sib.ch/announce/
CC   or send an email to license@isb-sib.ch).
CC   ---------------------------------------------------------------------
SQ   MMVAWWSLFL YGLQVAAPAL AATPADWRSQ SIYFLLTDRF ARTDGSTTAT CNTADQKYCG GTWQGIIDKL
DYIQGMGFTA IWITPVTAQL PQTTAYGDAY HGYWQQDIYS LNENYGTADD LKALSSALHE RGMYLMVDVV
ANHMGYDGAG SSVDYSVFKP FSSQDYFHPF CFIQNYEDQT QVEDCWLGDN TVSLPDLDTT KDVVKNEWYD
WVGSLVSNYS IDGLRIDTVK HVQKDFWPGY NKAAGVYCIG EVLDGDPAYT CPYQNVMDGV LNYPIYYPLL
NAFKSTSGSM DDLYNMINTV KSDCPDSTLL GTFVENHDNP RFASYTNDIA LAKNVAAFII LNDGIPIIYA
GQEQHYAGGN DPANREATWL SGYPTDSELY KLIASANAIR NYAISKDTGF VTYKNWPIYK DDTTIAMRKG
TDGSQIVTIL SNKGASGDSY TLSLSGAGYT AGQQLTEVIG CTTVTVGSDG NVPVPMAGGL PRVLYPTEKL
AGSKICSSS
//
```

In the sample above, some the DE and SQ lines have been wrapped to fit the width of the page. In the data file, those would occur on a single line.

Assignment:

You will implement a system that maintains a database of amino acid sequences (proteins) stored in the format described above. There is no stated limit on the number of records that may be in the file, so all data structures must be fully dynamic.

Your system will build and maintain several in-memory index data structures to support the following operations:

- Retrieving protein records from the database file based on the accession code
- Retrieving protein records from the database file based on source organism
- Displaying the in-memory indices in a human-readable manner

You will implement a single C++ program to perform all system functions. Note that your program will not store any complete protein records in memory, aside from the particular one that is being used at any given time.

Program Invocation:

The program will take the names of three files from the command line, like this:

```
ProteinDB <database file name> <command script file name> <log file name>
```

If the database file is not found, open a new file using the given name, and begin execution with an empty database. Naturally, if the script file is not found, the program should log an error message and exit.

Data and File Structures:

There will be an initial database file, in the format described earlier. Adding a new protein record to the database requires updating the indexing data structures in memory as well as the initial database file on disk.

Each of the search keys is simply an ASCII string, and so the keys can be compared using the standard relational operators. The accession codes are unique (primary); that is, no two different proteins will have the same accession code. None of the other fields are guaranteed to be primary.

The index structure for the accession codes will be stored using a binary search tree. The index entries in the accession code index will store the accession code and the corresponding *record locator* of the corresponding protein record in the dB file. A *record locator* is a non-negative integer specifying the file offset at which the record begins.

The index for the source organisms will also be stored using a BST, but there may be many protein records that match a single organism. The source organism index will store index entries containing the organism name and a collection of corresponding primary accession codes (NOT record locators!). This means that retrievals based on the source organism name will require first searching the source organism index and then performing one or more searches of the primary accession code index.

Aside from where specific data structures are required, you may (and should) take advantage of any suitable STL component you like.

At the start of execution, your program should parse the database file and build both index structures. Each index object should have the ability to write a nicely-formatted display of itself to an output stream.

Other System Elements:

You are expected to apply the object-oriented design principles you were taught in the prerequisite courses when designing the system. The following discussion is intended only to provide food for thought. It is highly probable that there are other expected design elements that are not mentioned here.

There should be an overall controller that validates the command line arguments and manages the initialization of the indices. The controller should hand off execution to a command processor that manages retrieving commands from the script file, and making the necessary calls to other system elements, which will then carry out those commands. Index entries are objects. So are protein records. An index is more than just a naked container.

Command File:

The execution of the program will be driven by a script file. Lines beginning with a semicolon character (`';'`) are comments and should be ignored. Each non-comment line of the command file will specify one of the commands described below. Each line consists of a sequence of tokens, which will be separated by single tab characters. A newline character will immediately follow the final token on each line. The command file is guaranteed to conform to this specification, so you do not need to worry about error-checking when reading it. The following commands must be supported:

`display<tab><accession>`
: Log the complete protein record that has primary accession code `<accession>`.

`describe<tab><accession>`
: Log the description field in the protein record that has primary accession code `<accession>`. Do not log the complete record.

`organism<tab><species>`
: Log the accession code and file offset for every protein record that includes the organism name given in `<species>`. At the end of the list, log the number of matching protein records that were found. Do not log the records themselves.

`debug<tab>[accession | organism]`
: Log the contents of the specified structure in a fashion that makes the internal structure and contents of the index clear. It is not necessary to be overly verbose here, but it would be useful to include information like key values, file offsets, and record lengths where appropriate.

`exit<tab>`
: Terminate program execution.

For up to 10% extra credit:

`remove<tab><accession>`
: Remove the corresponding entry from the accession code index. Remove the accession code from any relevant entries in the organism index. Do not modify the database file.

A sample command script is included in Figure 2 below. As a general rule, every command should result in some output. In particular, error messages should be logged if searches yield no protein records. Note: the scripts may contain `remove` commands; if you don't implement support for `remove` commands, you must at least make sure that your program handles them gracefully when they occur in the scripts.

Instrumentation:

Each index (or its aggregated container) must be instrumented so that it logs information about each search it performs. The information should display each index record that is accessed during the index search, and should be written to the log file. Note that this information will be used to assess the correctness and performance of your data structures, so if you don't do this you should expect major deductions.

Figure 2 Sample Command Script:

```
; Test script for protein database project
; Display initial indices:
debug    accession
debug    organism
;
; Describe a few records:
describe O58489
describe Q57577
;
; Display a few records:
display  O58489
display  Q57577
;
; Remove a few sequences:
remove   O58489
remove   O27743
;
; Find a few organisms:
organism Methanopyrus kandleri
organism Aeropyrum pernix
;
; Quit
exit
```

File Navigation:

Performing some of the search functions in this assignment requires being able to move to an arbitrary location in the database file. This is easy to do if you make use of the appropriate features of C++ stream objects.

Every input stream maintains an internal data member that stores the current position of the input pointer, called the *get pointer*, within the file. The value of the get pointer is just the offset in bytes of the current position from the beginning of the file. Bytes are numbered sequentially, starting at zero, just like the cells of an array. The get pointer moves forward automatically when the usual input operations are performed on the stream object.

The current position of the get pointer can be obtained by calling the member function `tellg()` on the stream object. The get pointer can be moved to an arbitrary offset within the file by calling the member function `seekg()` and passing it the specified file offset.

Log File Description:

Since this assignment will be graded by TA, rather than the Curator, the format of the output is left up to you. Of course, your output should be clear, concise, well labeled, and correct.

The remainder of the log file output should come directly from your processing of the command file. You are required to echo each command that you process to the log file so that it's easy to determine which command each section of your output corresponds to. Each command should be numbered, starting with 1, and the output from each command should be well formatted, and delimited from the output resulting from processing other commands. A complete sample log will be posted shortly on the course website.

Submitting Your Program:

You will submit this assignment to the Curator System (read the *Student Guide*), where it will be archived for grading by a TA.

For this assignment, you must submit a zip file containing all the source code files for your implementation (i.e., header files and cpp files). Submit only the header and cpp files. Submit nothing else.

In order to correct submission errors and late-breaking implementation errors, you will be allowed up to five submissions for this assignment. You may choose which one will be evaluated at your demo, but we will evaluate only one submission.

The *Student Guide* and link to the submission client can be found at: `http://www.cs.vt.edu/curator/`

Evaluation:

Note that the evaluation of your project will depend substantially on the quality of your code and documentation. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

You will generally not be allowed to make any changes to your submitted code during a project demo. If the TA determines that it is not possible to fairly evaluate your submission without allowing you to make changes, he will document the changes that you make, and I will assess a penalty for those changes. The penalty will never be less than the equivalent of a one-day late penalty, and will usually be more.

Pedagogic points:

The goals of this assignment include, but are not limited to:

- creation of a sensible OO design for the overall system, including the identification of a number of useful classes not explicitly named in this specification
- implementation of such an OO design into a working system
- incremental testing of the basic components of the system in isolation
- satisfaction when the entire system comes together in good working order

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Submitting Assignments page of the course website.